**Stop Hardcoding Your Unit Tests**

**A guide to property-based testing in Python using Hypothesis**



Photo by [Jonatan Pie](https://unsplash.com/@r3dmax?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

The work of a data scientist has many facets: they have to understand business problems, know their algorithms and statistics, and write well-documented and tested code, among other things. This is a demanding task, and very often you have to make a decision on what to prioritize, given your limited time capacity.

From my experience, the coding part gets usually neglected a bit — code is prototyped in some notebook, and then at some point, it runs *good enough*, and that’s it. It also happens to me. And I can even understand it: documenting things is just no fun, but without it, nobody (including yourself) will be able to understand your code and it might get rewritten in the future, even if you write the best piece of code ever. So, even if code documentation is not the focus of this article, still **document your code**! What I am more interested in now is the **code testing** bit.

**Code Testing**

You should always test your code to gain confidence that it really does what you expect it to.

Without proper testing, your code might do the wrong thing and your analyses will give misleading results.

So, how can you test your code? Chances are that you have used [**unittest**](https://docs.python.org/3/library/unittest.html) or [**pytest**](https://docs.pytest.org/) to write some basic tests. Even if not, just read on as I will explain how it works using a small example. We can then compare this *conventional* method of testing (**example-based testing**) to something called **property-based testing**, which is the focus of the article.

As a running example let us use the implementation of a **temporal train-test split**. It should do the following: Given…

* a pandas data frame data containing a column called *date*
* a string variable split\_date

output two pandas data frames,

* a **train** data frame containing the part of data where the values in the *date* column come **before** split\_date
* a **test** data frame containing the part of data where the values in the *date* column come **after** split\_date .

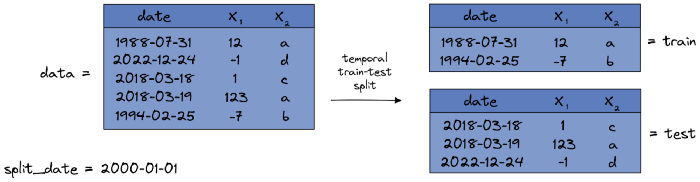


Image by the author.

Simple enough, right? Now, before we can test the function, we have to implement it first:

def temporal\_train\_test\_split(data, split\_date):  
 train = data.query("date < @split\_date")  
 test = data.query("date > @split\_date")  
  
 return train, test

***Note:*** *There is an error in the code, but let’s pretend that we overlooked it.*

**Example-Based Testing**

In order to test the code you would usually hardcode some input data, apply the function to it and check it with a hardcoded expected output. You could test the example from the image above like this:

import pandas as pd  
  
  
#function we want to test  
def temporal\_train\_test\_split(data, split\_date):  
 train = data.query("date < @split\_date")  
 test = data.query("date > @split\_date")  
  
 return train, test  
  
# testing function  
def test\_temporal\_train\_test\_split():  
 data = pd.DataFrame(  
 {  
 "date": [  
 "1988-07-31",  
 "2022-12-24",  
 "2018-03-18",  
 "2018-03-19",  
 "1994-02-25",  
 ],  
 "x1": [12, -1, 1, 123, -7],  
 "x2": ["a", "d", "c", "a", "b"],  
 }  
 )  
  
 split\_date = "2000-01-01"  
  
 expected\_output\_train = pd.DataFrame(  
 {  
 "date": ["1988-07-31", "1994-02-25"],  
 "x1": [12, -7],  
 "x2": ["a", "b"]  
 },  
 index=[0, 4],  
 )  
  
 expected\_output\_test = pd.DataFrame(  
 {  
 "date": ["2022-12-24", "2018-03-18", "2018-03-19"],  
 "x1": [-1, 1, 123],  
 "x2": ["d", "c", "a"],  
 },  
 index=[1, 2, 3],  
 )  
  
 train, test = temporal\_train\_test\_split(data, split\_date)  
  
 assert train.equals(expected\_output\_train)  
 assert test.equals(expected\_output\_test)

Save the above code in a file such as test\_temporal\_train\_test\_split.py and execute it via pytest test\_temporal\_train\_test\_split.py after you installed pytest via pip install pytest. pytest will then run the test and should report that it succeeded. Nice, so we can be a little bit more confident that our code works as intended!

However, you probably noticed that this single test does not cover much of the variety of inputs this function can potentially receive. It is only a single case that happens to work. And probably you will also miss some **edge cases** where your code might break.

The next best thing you can do is to define some more input/output pairs to increase the coverage a bit but this is really no fun because you have to hardcode a lot of data frames. And because example-based testing is so tedious to do, usually you see only a handful of hardcoded examples at most if you look into test files.

Don’t get me wrong, even a small test like this is better than no test at all but still, we can do better using **property-based testing**.

**Property-Based Testing**

If you want to do property-based testing, first you have to take some minutes and think about what the **properties** are that you want to see from the output. Sometimes this might be difficult to do, but in our case, it is quite easy. The outputs train and test should have the following properties:

1. The *date* column in train should always be less or equal to split\_date and the *date* column in test should always be greater than split\_date .
2. If you concatenate train and test , you should receive the input data frame back, i.e. no rows or columns get added or lost, and no cell gets changed.

This is really the essence of the temporal train-test split, the defining property, and we can directly test it.

def test\_temporal\_train\_test\_split\_property():  
 data = pd.DataFrame(  
 {  
 "date": [  
 "1988-07-31",  
 "2022-12-24",  
 "2018-03-18",  
 "2018-03-19",  
 "1994-02-25",  
 ],  
 "x1": [12, -1, 1, 123, -7],  
 "x2": ["a", "d", "c", "a", "b"],  
 }  
 )  
  
 split\_date = "2000-01-01"  
  
 train, test = temporal\_train\_test\_split(data, split\_date)  
 concatenated = (  
 pd.concat([train, test])  
 .sort\_values(["date", "x1", "x2"]) # see note below  
 .reset\_index(drop=True)  
 )  
 sorted\_input = data.sort\_values(["date", "x1", "x2"]).reset\_index(drop=True) # see note below  
  
 assert (train["date"] <= split\_date).all() # 1st property  
 assert (test["date"] > split\_date).all() # 1st property  
 assert concatenated.equals(sorted\_input) # 2nd property

***Note:*** *You have to sort the data frames because splitting a data frame into train and test and putting them together again might change the order of the rows. But even in this case the check should pass, order doesn’t matter.*

This is much better already since you **don’t have to hardcode the outputs** anymore! You only have to define some input data and let the properties take care of the rest. This even opens the door to another trick:

Wouldn’t it be great to generate a bunch of random inputs and then run the property checks?

This idea is quite simple and I bet that you could come out with some custom code to do just that. However, I’d rather want to show you a neat library that helps you implement this idea, even with a cherry on top as you will see! It is called [**Hypothesis**](https://hypothesis.readthedocs.io/), and I will show you how it works in the rest of the article.

**Property-Based Testing on Steroids With Hypothesis**

First, install the library with a simple pip install hypothesis . Before we write our test, let’s play around with it a bit. One thing hypothesis is great is generating random data. First, let us import the library:

import hypothesis.strategies as st

**Getting Started**

And now let’s do a very simple task: **“Generate random integers!”**.

integer\_strategy = st.integers()  
  
for \_ in range(5):  
 print(integer\_strategy.example())  
  
# Example output:  
# 26239  
# -32170  
# 8226  
# 12448  
# -25828

This should give you some random integers. Ok, but [**numpy**](https://numpy.org/) can do the same, so why bother? Take a look at another example: **“Generate lists of random integers!”**.

integer\_list\_strategy = st.lists(st.integers())  
  
for \_ in range(5):  
 print(integer\_list\_strategy.example())  
  
# Example output:   
# [-14, -2189, 9898, 116]  
# [115252802955829576656830209704323089026, 12850, -22, -23389, -37044854417799513209994526228023296414, 9033, -25431, 111, 1650017586, 2100275240795033860, 14027, 9549, 119, 32276, 3287]  
# [867485840, -16288]  
# [867485840, -16288]  
# [23623, 18045420794467201802863702411254425247, 11413941429584508497211673000716218542, -35326783386759048949361175218368769135, 25, 18663, 85, 29311, -54]

This is a bit more interesting. Without much effort, we could produce some random lists that we could use to test sorting algorithms.

We can combine more of these *strategies* to do even crazier stuff, such as: **“Generate lists of tuples containing random integers and boolean!”**.

strategy = st.lists(st.tuples(st.integers(), st.booleans()))  
  
for \_ in range(5):  
 print(strategy.example())  
  
# Example output:  
# [(-28942, True), (39, True), (2034980965, True), (-633849778, False), (-111, False), (-25, True), (15592, True), (-6976, False), (-29086, True), (20529, False), (-28691, True), (-6358763519719057102, False)]  
# [(-83, False), (0, True), (16, False), (-21, True), (32707, True), (-45239080, True), (115, False), (567947076, True), (-7363, False)]  
# [(-100, False), (-14515, True), (32539, False), (-22134, True), (-1419424594, False), (-21631, False)]  
# [(3401224866052712356, True), (-663846088058567152, True), (26848, False), (71, True), (-4004, True), (-84, True), (5403, True), (31368, False)]  
# [(21237, False), (-29568, True), (978088832, False), (-1095376597, True)]

Sky is the limit! And you can pass keywords to all of the strategies, for example, to limit the size of the integers or list lengths.

strategy = st.lists(  
 st.tuples(  
 st.integers(min\_value=0, max\_value=5),  
 st.booleans()  
 ),  
 min\_size=2, max\_size=4  
 )  
  
for \_ in range(5):  
 print(strategy.example())  
  
# Example output:  
# [(0, True), (0, True), (0, True), (0, True)]  
# [(5, False), (3, True), (2, True), (5, True)]  
# [(2, True), (4, True), (1, True), (2, False)]  
# [(2, False), (2, True), (2, True), (2, True)]  
# [(1, False), (5, False), (2, True)]

There are so many more strategies, and I encourage you you [read the Hypothesis docs](https://hypothesis.readthedocs.io/en/latest/data.html) to find out more about them, but let us focus on our use case now.

**Testing Our Code With Hypothesis**

As a reminder, our temporal train-test split has two inputs:

* a pandas data frame data containing a column called *date*
* a string variable split\_date

Let us first deal with generating random dates since this is easier. You have probably guessed that you can do it like this:

date\_strategy = st.dates()  
  
for \_ in range(5):  
 print(date\_strategy.example())  
  
# Example output:  
# 1479-03-03  
# 3285-02-06  
# 0715-06-28  
# 6354-02-18  
# 9276-08-08

This strategy produces Python date objects, but since we expect our function to take a date in form of a **string**, we can map it to one via

st.dates().map(lambda d: d.strftime("%Y-%m-%d"))

The data frame is a more complex data type but Hypothesis got us covered here. It offers several ways to define data frames, but I’ll show you the most general one using the composite decorator.

The following code snippet

1. creates a random number of rows, then
2. creates this amount of rows containing a date, integer, and one of the letters a, b, c, d, and then
3. makes a pandas data frame out of it and outputs it.

@st.composite  
def random\_dataframe\_with\_a\_date\_column\_strategy(draw):  
 n\_rows = draw(st.integers(min\_value=0, max\_value=100))  
  
 rows = [  
 (  
 draw(st.dates().map(lambda d: d.strftime("%Y-%m-%d"))),  
 draw(st.integers()),  
 draw(st.sampled\_from(list("abcd"))),  
 )  
 for \_ in range(n\_rows)  
 ]  
 data = pd.DataFrame(rows, columns=["date", "x1", "x2"])  
  
 return data

Note the different occurrences of draw there. You need this because functions like range want proper integers, for example. As with any other strategy, you can do

random\_dataframe\_with\_a\_date\_column\_strategy().example()

now to receive a random example.

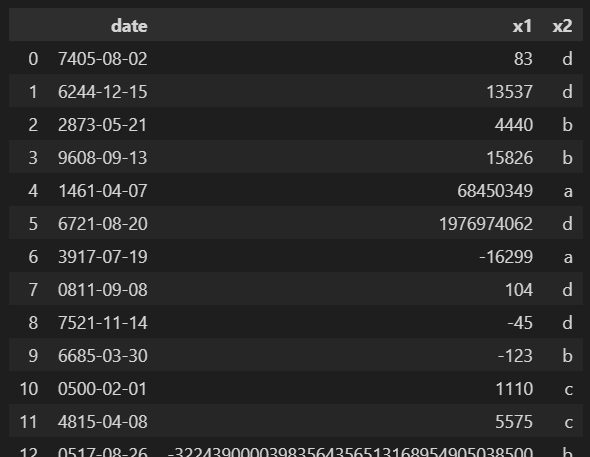


Image by the author.

And of course, you can make even more general data frames. We just vary the number of rows here, but you can also

* vary the number of columns
* vary the data types
* vary the column names

among other things. But let’s first understand this more specific strategy.

Great, we can generate random input data now! The only thing that remains is to feed it to our testing function now in a proper way. Luckily, Hypothesis makes this very easy. Just create another file called something like test\_temporal\_train\_test\_split\_hypothesis.py and paste the following:

import pandas as pd  
from hypothesis import given, note  
import hypothesis.strategies as st  
  
# Function to test  
def temporal\_train\_test\_split(data, split\_date):  
 train = data.query("date < @split\_date")  
 test = data.query("date > @split\_date")  
  
 return train, test  
  
  
# Strategies  
split\_date\_strategy = st.dates().map(lambda d: d.strftime("%Y-%m-%d"))  
  
  
@st.composite  
def random\_dataframe\_with\_a\_date\_column\_strategy(draw):  
 n\_rows = draw(st.integers(min\_value=0, max\_value=100))  
  
 rows = [  
 (  
 draw(st.dates().map(lambda d: d.strftime("%Y-%m-%d"))),  
 draw(st.integers()),  
 draw(st.sampled\_from(list("abcd"))),  
 )  
 for \_ in range(n\_rows)  
 ]  
 data = pd.DataFrame(rows, columns=["date", "x1", "x2"])  
  
 return data  
  
# The actual test  
@given(  
 data=random\_dataframe\_with\_a\_date\_column\_strategy(), split\_date=split\_date\_strategy  
)  
def test\_temporal\_train\_test\_split(data, split\_date):  
 note(data) # basically a print function if a test fails  
 note(split\_date) # basically a print function if a test fails  
  
 train, test = temporal\_train\_test\_split(data, split\_date)  
 concatenated = (  
 pd.concat([train, test])  
 .sort\_values(["date", "x1", "x2"])  
 .reset\_index(drop=True)  
 )  
 sorted\_input = data.sort\_values(["date", "x1", "x2"]).reset\_index(drop=True)  
  
 assert (train["date"] <= split\_date).all()  
 assert (test["date"] > split\_date).all()  
 assert concatenated.equals(sorted\_input)

You know most things about this code already. First, we define the function that we want to test as well as the strategies for the random inputs. Then we can conduct the test using the given decorator from Hypothesis. That’s it already! We can let it run again via

pytest test\_temporal\_train\_test\_split\_hypothesis.py

This will by default create 100 random inputs and check the properties.

Probably you will see something like this:

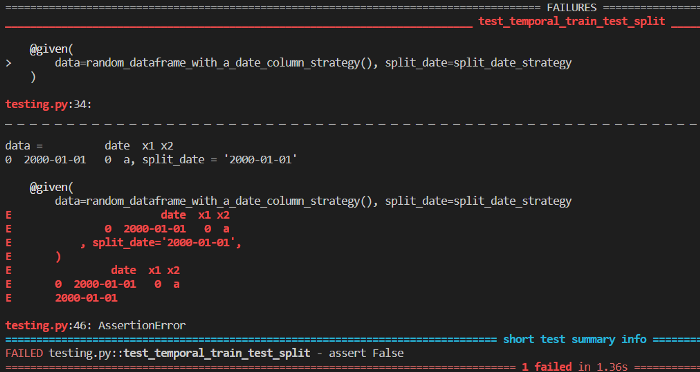


Image by the author.

Oops, our function still contains an error! And Hypothesis is nice enough to tell us which inputs create the error because we used note(data) and note(split\_date) in the code.

**Shrinking Examples**

Another remarkable thing is that this example is nice and short, so we can directly guess what the error might be. **This is by no means a coincidence.** Whenever Hypothesis finds an example that breaks your code — maybe in our case a data frame containing 98 rows — it tries to simplify this example such that it **still breaks your code**, but is **smaller** in a sense.

This process is called **shrinking**, and it makes it easier for a human to get to the source of the error. Shrinking usually feels natural:

* shrinking numbers means getting them closer to zero
* shrinking lists mean making them shorter
* shrinking strings means making them shorter
* …

So, let’s use the error message to think about what went wrong!

**Fixing the Error**

We can easily see that the example that breaks our code has a split\_date that is equal to the date value in data , which really looks suspicious. Probably our code doesn’t work in this case. Let’s take a look at it again:

def temporal\_train\_test\_split(data, split\_date):  
 train = data.query("date < @split\_date")  
 test = data.query("date > @split\_date")  
  
 return train, test

Yes, it makes sense. We use **<** and **>**, but what happens with the dates that are equal to split\_date ? They just get dropped in our version. 😖 We can fix this with a simple

def temporal\_train\_test\_split(data, split\_date):  
 train = data.query("date <= @split\_date") # fixed  
 test = data.query("date > @split\_date")  
  
 return train, test

If we let it run again, the test will pass now, awesome!

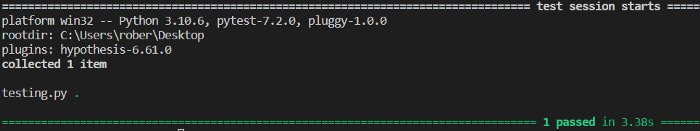


Image by the author.

And again, I want to stress that even if it looks like a single test is running, actually **100 tests ran**.

**More Features**

You can also change this number using the setting decorator, for example like this:

from hypothesis import settings  
  
@given(  
 data=random\_dataframe\_with\_a\_date\_column\_strategy(), split\_date=split\_date\_strategy  
)  
@settings(max\_examples=200) # number of random examples  
def test\_temporal\_train\_test\_split(data, split\_date):  
 [...]

Also, sometimes you want to make sure that some specific example that is very important to you gets covered, and you don’t want to leave it to chance. In this case, you can use the example decorator like this:

from hypothesis import example  
  
@given(  
 data=random\_dataframe\_with\_a\_date\_column\_strategy(), split\_date=split\_date\_strategy  
)  
@example(data=pd.DataFrame(None, columns=["date", "x1", "x2"]), split\_date="9999-12-31") # this example is always covered  
def test\_temporal\_train\_test\_split(data, split\_date):  
 [...]

Now you know the basics about property-based testing with Hypothesis and you can try to apply it in your projects!

**Conclusion**

Testing your code is a tedious task, but still you have to do it to spot mistakes that might ruin your results. You can test your code by providing input/output pairs (examples), but people tend to hardcode only a handful of examples at most. This leads to a small coverage and your code might still not work for a large fraction of inputs, or for some edge cases that might be important.

Property-based testing is a convenient way to increase coverage. It tests random inputs and the Hypothesis library can even provide you with simple examples that break your code, so bug fixing becomes much easier. The downside is that you have to come up with **meaningful properties**. Sometimes this is easy, as for our temporal train-test split example. Sorting numbers is another great use case for property-based testing. You only have to check that

1. the numbers in the output are in increasing order and
2. that no numbers get added or dropped.

**🚀 Try implementing a sorting algorithm of your choice and test it using property-based testing!**

Every problem in the complexity class NP is actually a good candidate for property-based testing since verifying solutions can be done efficiently.

But sometimes it might be hard to make up good properties. Sometimes you also only find a few, but not all defining properties. In this case, you can still check all the properties you can think of and provide some manual examples as in conventional example-based testing.

Nothing keeps you from using both approaches together! Very often, this is even a great thing to do.

I hope that you learned something new, interesting, and useful today. Thanks for reading!

**As the last point, if you**

1. **want to support me in writing more about machine learning and**
2. **plan to get a Medium subscription anyway,**

**why not do it** [**via this link**](https://dr-robert-kuebler.medium.com/membership)**? This would help me a lot! 😊**

*To be transparent, the price for you does not change, but about half of the subscription fees go directly to me.*